Distributed Algorithms

Models of Distributed Systems

Models

- What is a model?
  - An abstraction of the relevant properties of a system
- Why construct or learn a model?
  - Real world is complex, a model makes assumptions and simplifications
  - Reason about realities in the model
  - Helps us tackle the complexities
  - The model and its properties are expressed in precise mathematical symbols and relationships

Modeling

- What can modeling do for us?
  - Useful when *solving* problems (e.g. designing an algorithm)
  - When *predicting* behavior (e.g. cost in number of messages)
  - When *evaluating* and *verifying* a solution (e.g. simulation)
- Very important skill

Modeling

- Different types of models:
  - Continuous models
    - Often described by differential equations involving variables which take real (continuous) values
  - Discrete event models
    - Often described by state transition systems: system evolves, moving from one state to another at discrete time steps
- This course: *a model of distributed computing (discrete)*

Models of distributed computing

- Biggest challenge when modelling is to choose the *right level of abstraction*!

- The model should be powerful enough to construct *impossibility proofs*
  - A statement about all possible algorithms in a system
- Our model should therefore be:
  - *Precise*: explain all relevant properties
  - *Concise*: explain a class of distributed systems compactly

Input/output Automata

Input/Output Automata

- General mathematical modeling framework for reactive system components

- Designed for describing systems in a modular way

  - Supports description of individual system components, and how they compose to yield a larger system

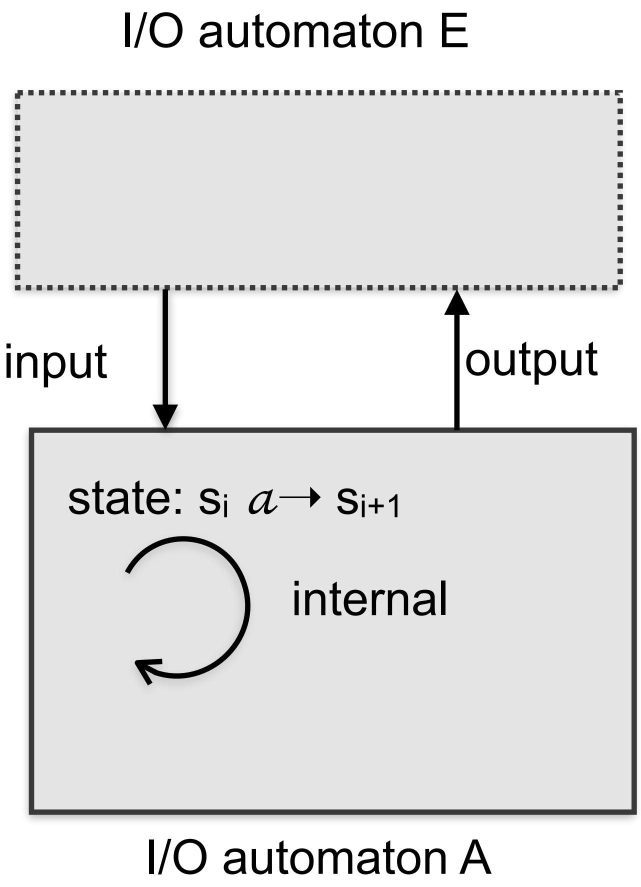
  - Supports description of systems at different levels of abstraction

I/O Automata

- A distributed algorithm (system) is specified as an Input/Output automaton
- I/O automata models concurrent interacting components
  - Suitable for components that interact asynchronously
- Each I/O automaton is a reactive state-machine:
  - Interacts with environment through actions
  - Makes transitions (state, action, state)
    - $\langle s_i , a, s_{i+1} \rangle$
- Actions, Events
  - Input, Output, Internal

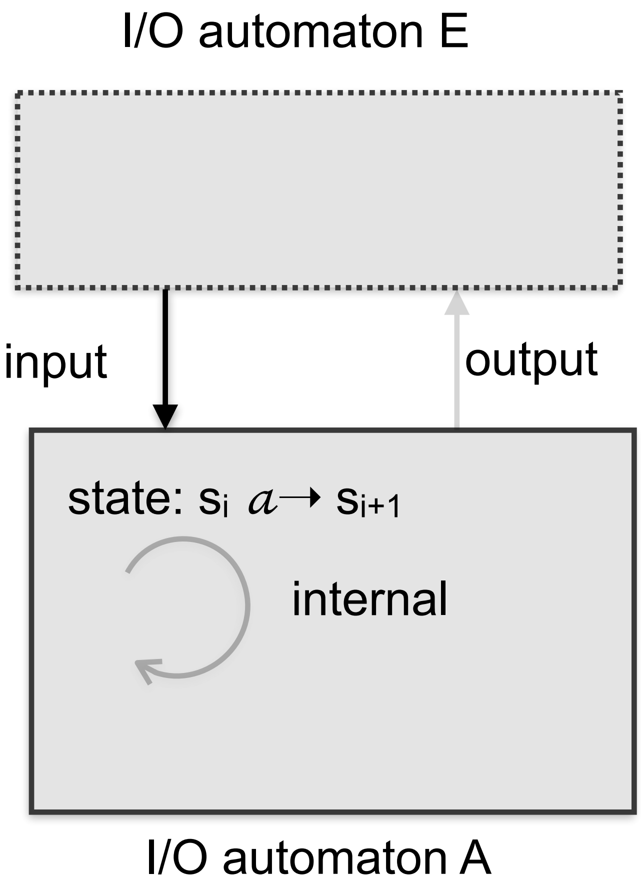
I/O Automata

- A distributed algorithm (system) is specified as an Input/Output automaton
- I/O automata models concurrent interacting components
  - Suitable for components that interact asynchronously
- Each I/O automaton is a reactive state-machine:
  - Interacts with environment through actions
  - Makes transitions (state, action, state)
    - $\langle s_i , a, s_{i+1} \rangle$
- Actions, Events (occurrence of action)
  - Input, Output, Internal



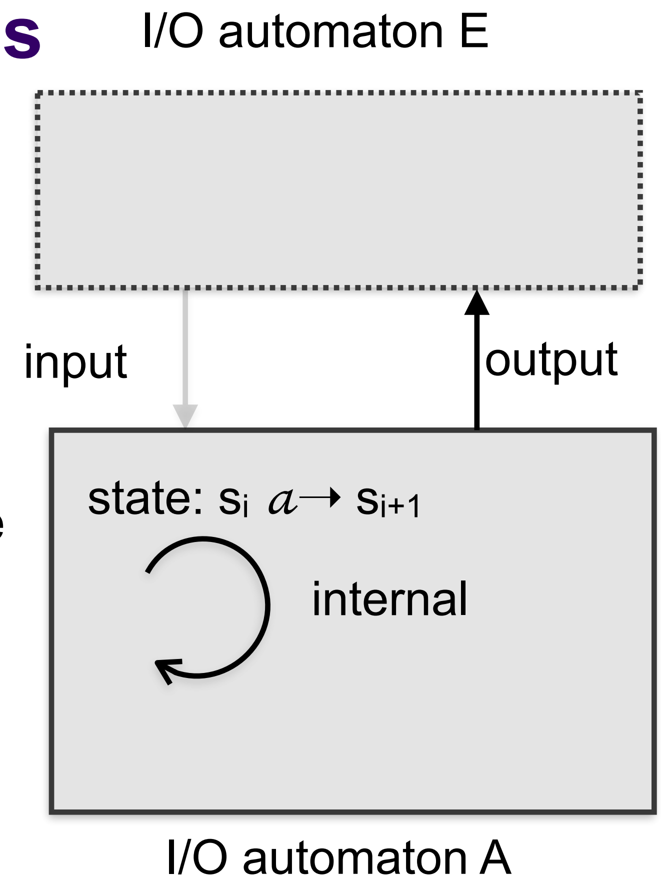
Input Actions

- Actions are named a_1, a_2, …
- Input of automaton A
  - Always enabled
  - Environment E with output action a can always invoke input action a of Automaton A
  - E and A both make a simultaneous transition
- A does not control its input action a



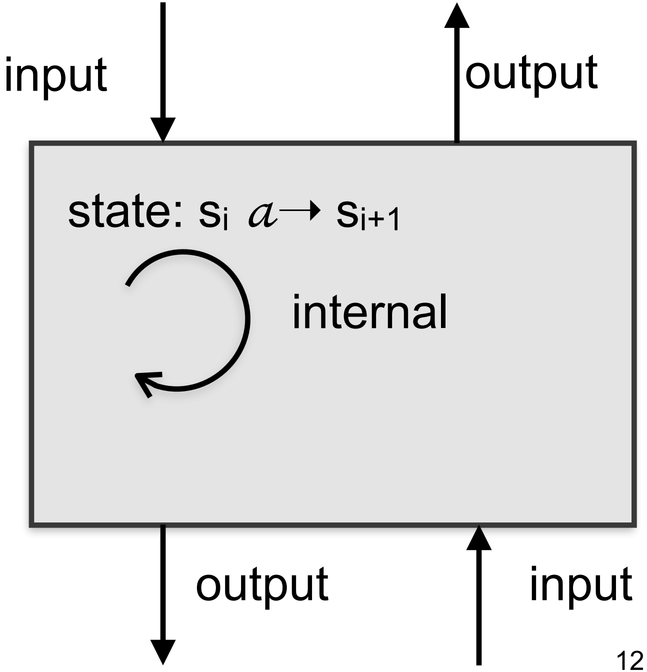
Internal and Output Actions

- Actions are named a_1, a_2, …
- Output, Internal actions of automaton A
  - Conditioned on A's state
  - Can be blocked until the condition is true
- A controls its internal and output actions



Input/Output Automaton

- Labeled State transition system
  - Transitions labeled by actions
- Actions classified as input, output, internal
  - Input, output are external
  - Output, internal are locally controlled.



Signature, formally

- Signature S
  - *in(S)*, *out(S)*, and *int(S)*
  - Input, output and internal actions
- in(S) ∪ out(S) ∪ int(S) disjoint
- External actions ext(S)
  - in(S) U out(S)
- Locally controlled actions local(S)
  - out(S) U int(S)

Automaton A is a labeled transition System

- states(A)
  - a (not necessarily finite) set of states
- start(A)
  - a nonempty subset of states(A)
- trans(A) a state-transition relation
  - trans(A) $\subseteq$ states(A) × acts(sig(A)) × states(A)
- For every state s and every input action a, there is a transition (s, a, s') $\in$ trans(A)
- Tasks: local actions are partitions into groups

Executions

- Running an I/O automata generate executions
- Execution
  - A alternating sequence of state and actions
  - The execution of an action is called an event
- Fair Execution
  - Execution where internal and output actions are given infinitely many chances to run

Traces (behaviors)

- External actions
  - Input and output actions
- "Interesting" behavior of I/O automata is captured by its external actions during executions
- (Fair) Trace
  - Subsequence of fair execution that consists of external actions
- The set of all traces capture interesting behavior of I/O Automata

Automata A Solved P

- A problem P (a distributed abstraction) will be defined as a set of sequences of external actions

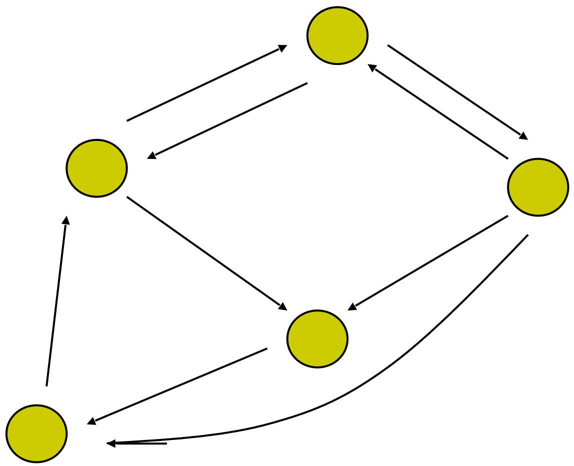
- Automaton A solves problem P
  - The set of fair behaviors of A is a subset of P

An asynchronous networked system

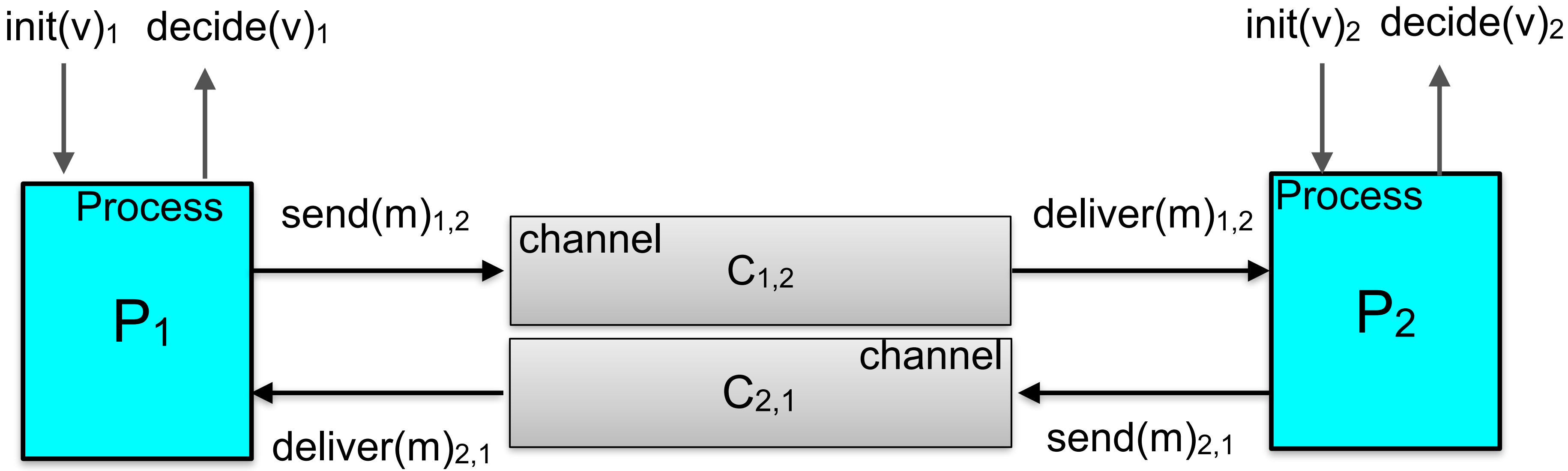
Example

an asynchronous networked system

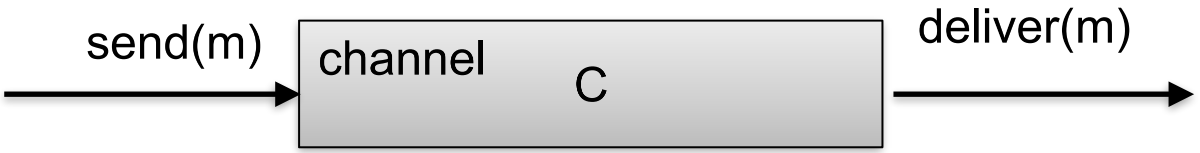
- An synchronous network
- Processes communicate via channels
- Processes and channels are
  - “Reactive” components that interact with their environments via input and output actions
  - modelled by I/O automata



Processes and channels

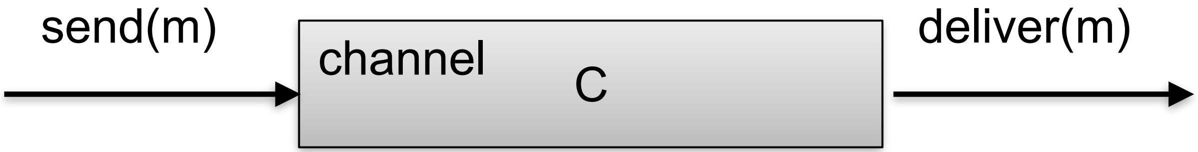


Example: Channel Automaton



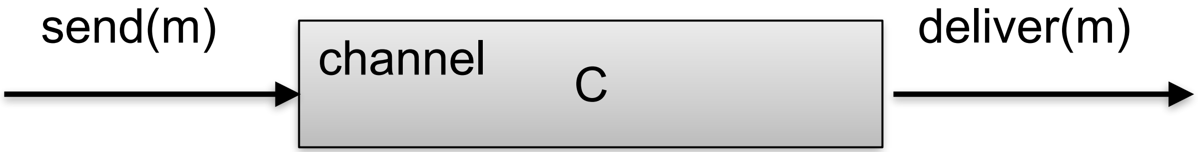
- Reliable unidirectional FIFO channel between two processes
  - Fix set of messages M
- Signature
  - Input actions: send(m), $m \in M$
  - Output actions: deliver(m), $m \in M$
  - No internal actions
- States
  - *queue*, a FIFO queue of elements of M, initially empty

Example: Channel Automaton



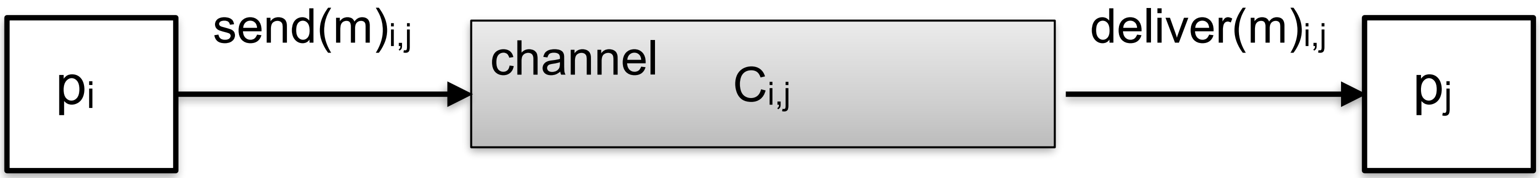
- Transitions
  - *send(m):*
    - *Effect: add m to(end of) queue*
  - *deliver(m):*
    - *precondition: m is first (head) in queue*
    - *Effect: remove m from queue*
- Tasks: all deliver actions is one task
- Transitions are described using “transition definitions”, which are little code fragments

Example: Channel Automaton



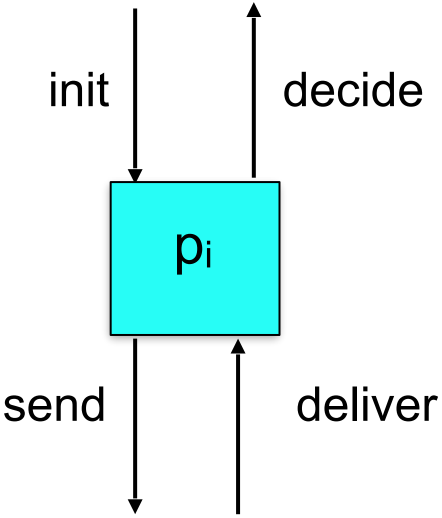
- Transitions
  - *send(m):*
    - *Effect: add m to(end of) queue*
  - *deliver(m):*
    - *precondition: m is first (head) in queue*
    - *Effect: remove m from queue*
- Transitions are described using “transition definitions”, which are little code fragments
- Each transition definition describes a set of transitions, for designated actions (grouped by type of action)

Example: Channel Automaton



- Add subscripts to indicate particular endpoints
- Here, the channel is used to connect processes i and j.
- Transitions
  - *$send(m)_{i,j}$:*
    - *Effect: add m to(end of) queue*
  - *$deliver(m)_{i,j}$:*
    - *precondition: m is first (head) in queue*
    - *Effect: remove m from queue*

A process

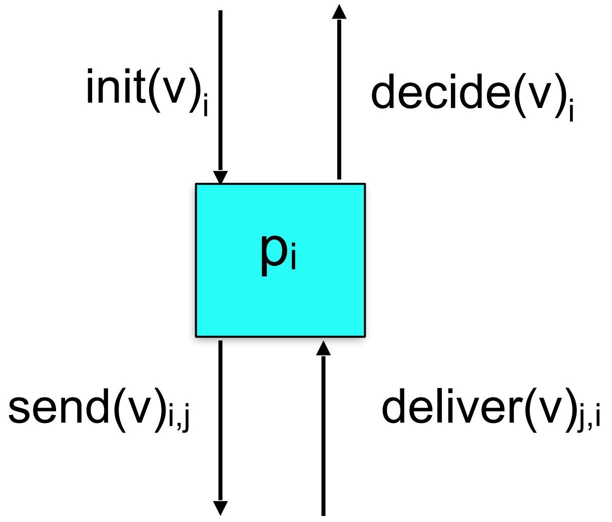


A simple agreement protocol

- Inputs arrive from the outside
- Process sends/receives values, collects vector of values, one for each process
- When vector is filled, outputs a decision obtained as a function **f** on the vector
- Can get new inputs, change values, send and output repeatedly
- Tasks for:
  - Sending to each individual neighbor
  - Outputting decisions

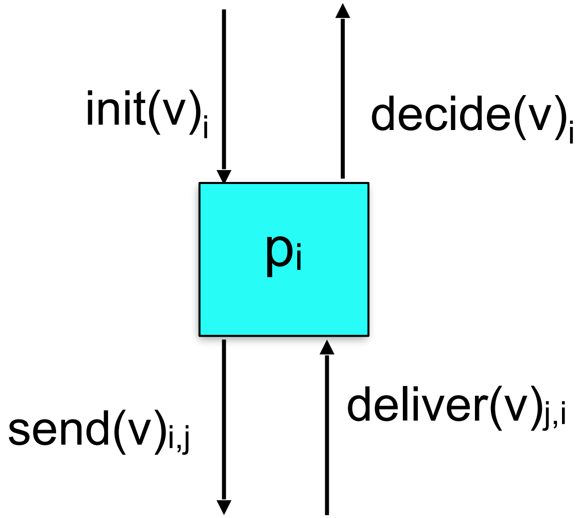
A process signature

- Input:
  - *$init(v)_i$, for $v \in$* V
  - *$deliver(v)_{j,i}$, $v \in$ V, $1 \leq j \leq n$, $j \neq i$*
- Output:
  - *$decide(v)_i$, $v \in$* V
  - *$send(v)_{i,j}$, $v \in$* V, 1 *$\leq j \leq n$, $j \neq i$*
- *States:*
  - *val,* a vector indexed by {1 , . . . , n} of elements in V U {⊥}, all initially ⊥ (**null**)



Transitions

- *$init(v)_i$* , $v \in V$: val(i) := v (input)
- *$deliver(v)_{j,i}$* , $v \in V$: val(j) := v (input)
- *$send(v)_{i,j}$* : (output)
  - *Precondition: val(i) = v*
  - *Effect: none*
- $decide(v)_i$: (output)
  - Precondition: for all $1 \leq j \leq n$: val(j) $\neq$ null
  - v = **f**(val(1),…,val(n))
  - Effect: none



Input/output Automata

Executions

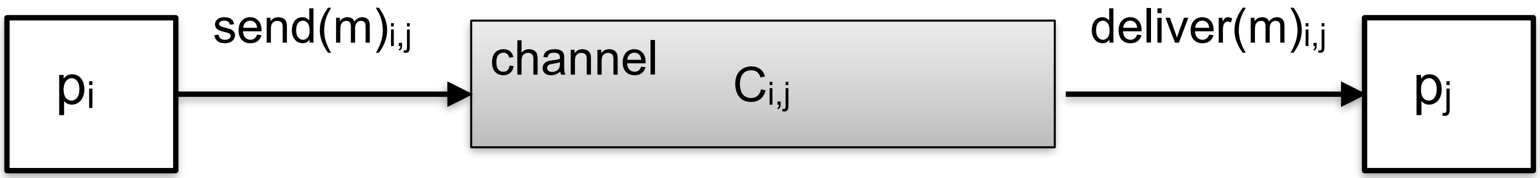
Remarks

- A step taken by automaton A is an element of trans(A)
- An action a is enabled in state s if trans(A) contains a step (s, a , s’) for some s’
- I/O automata are always input-enabled
  - Input actions are enabled in every state
  - An automaton cannot control its environment

Executions

- An I/O automaton executes as follows:
  - Start at some start state
  - Repeatedly take step from current state to new state.
- Formally, an execution is a finite or infinite sequence:
  - $s_0 \; a_1 \; s_1 \; a_2 \; s_2 \; a_3 \; s_3 \; a_4 \; s_4 \; a_5 \; s_5$ ... (if finite, ends in state)
  - s_0 is a start state
  - (s_i, a_{i+1}, s_{i+1}) is a step (i.e., in trans)

Executions: Channel Automaton



- Let M = {1,2}
- Three possible executions
- Any prefix of an execution is also an execution

1. [λ], send(1)$_{i,j}$, [1], deliver(1)$_{i,j}$, [λ], send(2)$_{i,j}$, [2], deliver(2)$_{i,j}$, [λ]
2. [λ], send(1)$_{i,j}$, [1], deliver(1)$_{i,j}$, [λ], send(2)$_{i,j}$, [2]
3. [λ], send(1)$_{i,j}$, [1], , send(1)$_{i,j}$, [11], , send(1)$_{i,j}$, [111], …

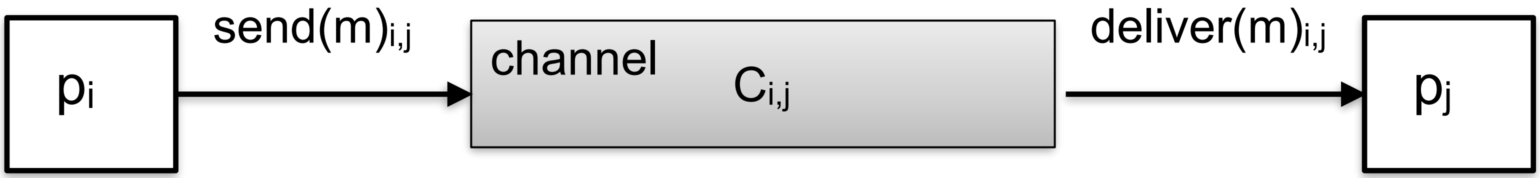
Execution Fragments

- An I/O automaton executes as follows:
  - Start at some start state.
  - Repeatedly take step from current state to new state.
- Formally, an execution fragment is a finite or infinite sequence:
  - $s_0 \ a_1 \ s_1 \ a_2 \ s_2 \ a_3 \ s_3 \ a_4 \ s_4 \ a_5 \ s_5$ ... (if finite, ends in state)
  - ~~s_0 is a start state~~
  - (s_i, a_{i+1}, s_{i+1}) is a step (i.e., in trans)

Traces

- Traces allows us to focus on the component's external behavior
- Useful for defining correctness of an algorithm
- A trace of an execution is the subsequence of external actions in the execution
  - No states, no internal actions
  - Denoted trace(E) where E is an execution
  - Models observable behavior of a component

Traces: Channel Automaton



- Let M = {1,2}
- Three possible executions and traces

1. [λ], $send(1)_{i,j}$, [1], $deliver(1)_{i,j}$, [λ], $send(2)_{i,j}$, [2], $deliver(2)_{i,j}$, [λ]
2. $send(1)_{i,j}$, $deliver(1)_{i,j}$, $send(2)_{i,j}$, $deliver(2)_{i,j}$
3. [λ], $send(1)_{i,j}$, [1], $deliver(1)_{i,j}$, [λ], $send(2)_{i,j}$, [2]
4. $send(1)_{i,j}$, $deliver(1)_{i,j}$, $send(2)_{i,j}$
5. [λ], $send(1)_{i,j}$, [1], , $send(1)_{i,j}$, [11], , $send(1)_{i,j}$, [111], …
6. $send(1)_{i,j}$, $send(1)_{i,j}$, $send(1)_{i,j}$, …

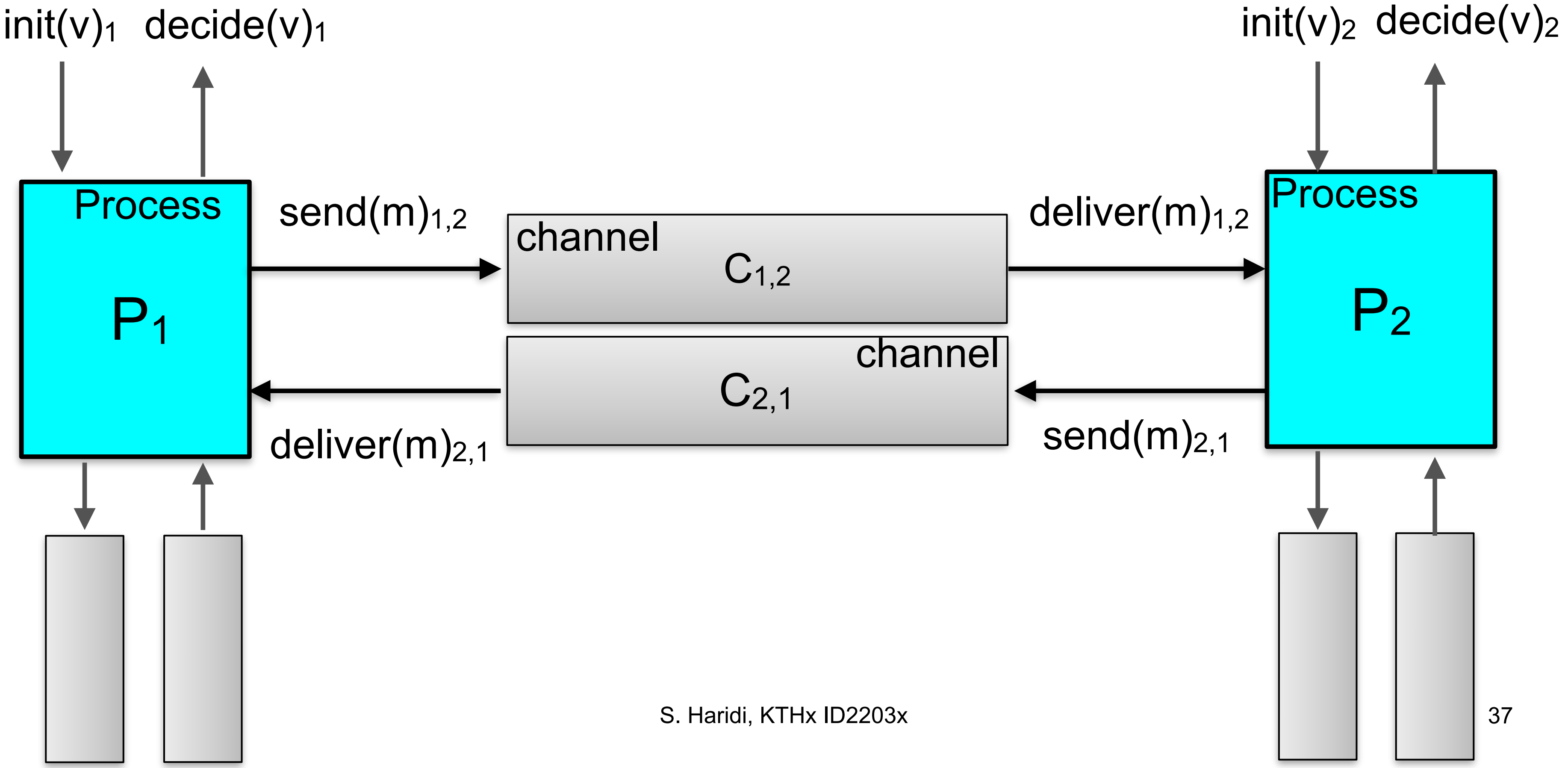
Input/output Automata

Operations on I/O automata

Composition

- Describes how systems are built out of components
- Main operations
  - Composition and hiding of actions
- Composition
  - Putting automata together to form a new automaton
  - Output action of one automaton with the matching input actions of the others
  - All components sharing the same action perform a step together (synchronize on actions)

Composition of channels and processes



Composition

- Composing multiple Automata $\{A_i, i \in I\}$, requires compatibility conditions
- for all $i, j \in I, i \neq j$
  - Internal actions are not shared
  - $int(A_i) \cap acts(A_j) = \varnothing$
  - Only **one** automaton controls each output
  - $out(A_i) \cap out(A_j) = \varnothing$
- However one output may be the input of many others

Composing Compatible Automata

- Composing Automata $A = \prod\{A_i, i \in I\}$
- Output actions of the components become output actions of the composition
- Internal actions of the components become internal actions of the composition
- Actions that are inputs to some components but outputs of none become input actions of the composition

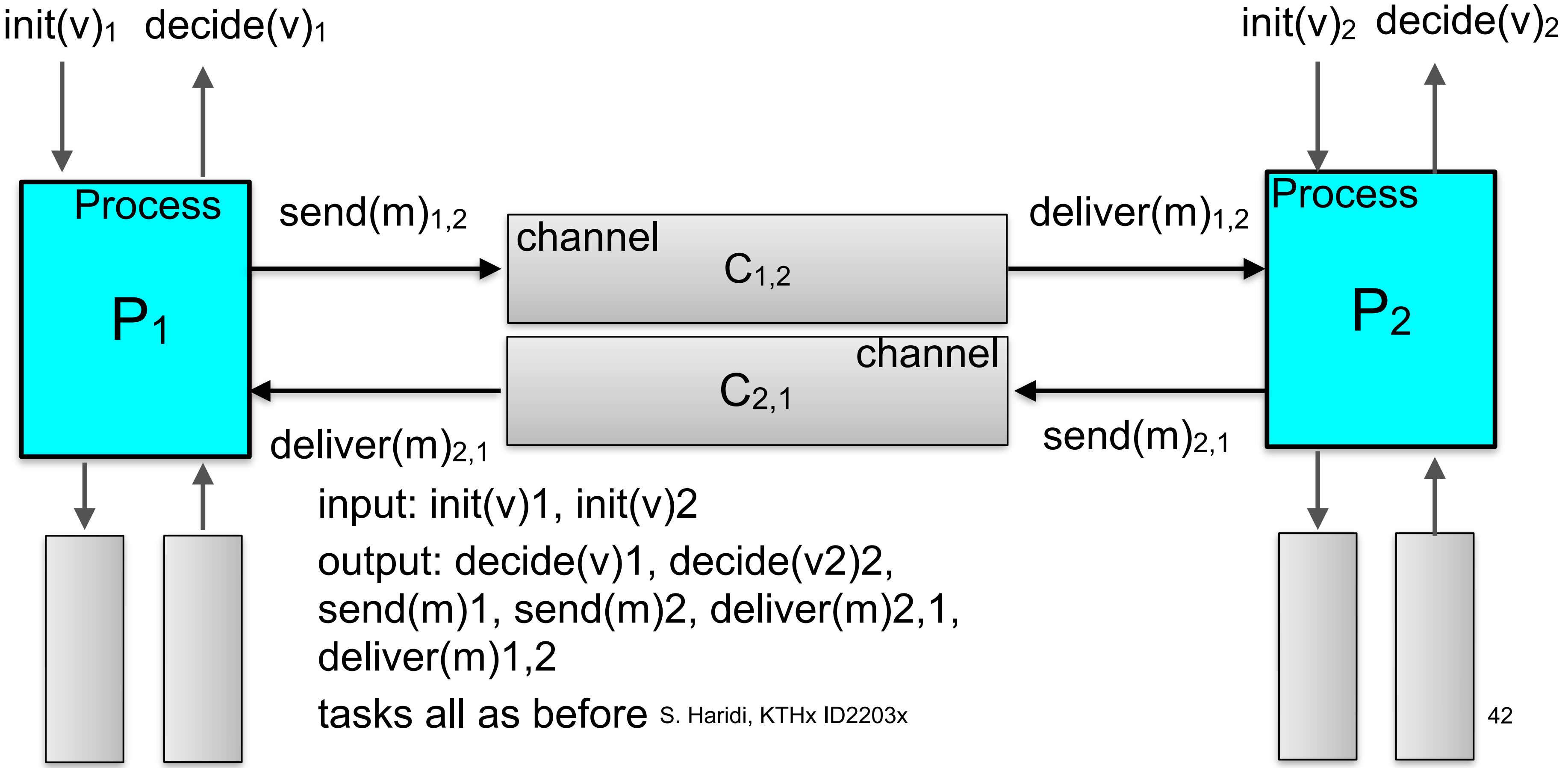
Composing Compatible Automata

- Composing Automata $A = \prod\{A_i, i \in I\}$
- Output actions of the components become output actions of the composition
  - $out(A) = \cup\{out(A_i), i \in I\}$
- Internal actions of the components become internal actions of the composition
  - $int(A) = \cup\{int(A_i), i \in I\}$
- Actions that are inputs to some components but outputs of none become input actions of the composition
  - $in(A) = \cup\{in(A_i), i \in I\} - out(A)$

Composing Compatible Automata

- Composing Automata $A = \prod\{A_i, i \in I\}$
- the states and start states of the composition are vectors of component states and start states, respectively, of the component automata
- $state(A) = \prod\{state(A_i), i \in I\}$
- $start(A) = \prod\{start(A_i), i \in I\}$
- The task partition of the composition's locally controlled actions is formed by taking the union of the components' task partitions
- $tasks(A) = \cup\{tasks(A_i), i \in I\}$

Composition of channels and processes



input: init(v)1, init(v)2

output: decide(v)1, decide(v2)2, send(m)1, send(m)2, deliver(m)2,1, deliver(m)1,2

tasks all as before

Transitions of Composed Automata

- Composing Automata $A = \prod\{A_i, i \in I\}$
- In a transition step, all the component automata that have a particular action a participate simultaneously in a
- Other component automata do nothing
- If a is output of automaton A1 and a in input of A2 and A3, but not sig(A4),
- A1, A2 and A3 take part and change their state
- $(s_1, s_2, s_3, s_4)\ a\ (s'_1, s'_2, s'_3, s_4)$

Transitions of Composed Automata

- Composing Automata $A = \prod\{A_i, i \in I\}$
- trans(A) is the set of triples (s, a, s') such that, the elements s'_i of vector s' is formed as follows:
  - for all $i \in I$ if $a \in acts(A_i)$, then $(s_i, a, s'_i) \in trans(A_i)$ otherwise $s_i = s'_i$
- The component states that change are those participating in the action a

Transitions of Composed Automata

- Composing Automata $A = \prod\{A_i, i \in I\}$
- Assume $(s, a, s') \in trans(A)$
  - if $a \in int(A)$ or $a \in in(A)$ then only one state component is changed in s to s’
  - if $a \in out(A)$ then multiple state components may change in s’, those A_i’s that participate in a

Hiding

- Turn output actions into internal actions
- Prevents outputs of composed automaton of further interaction with other automata under further composition
- Makes those output no longer included in traces
- S is a signature, $\Sigma \subseteq$ out(S), hide_{Σ} (S) is S' where
  - in(S') = in(S), out(S') = out(S) - Σ, int(S') = int(S) $\cup$ Σ
- hide_{Σ} (A) is an automaton A' whose signature is hide_{Σ} (sig(A))

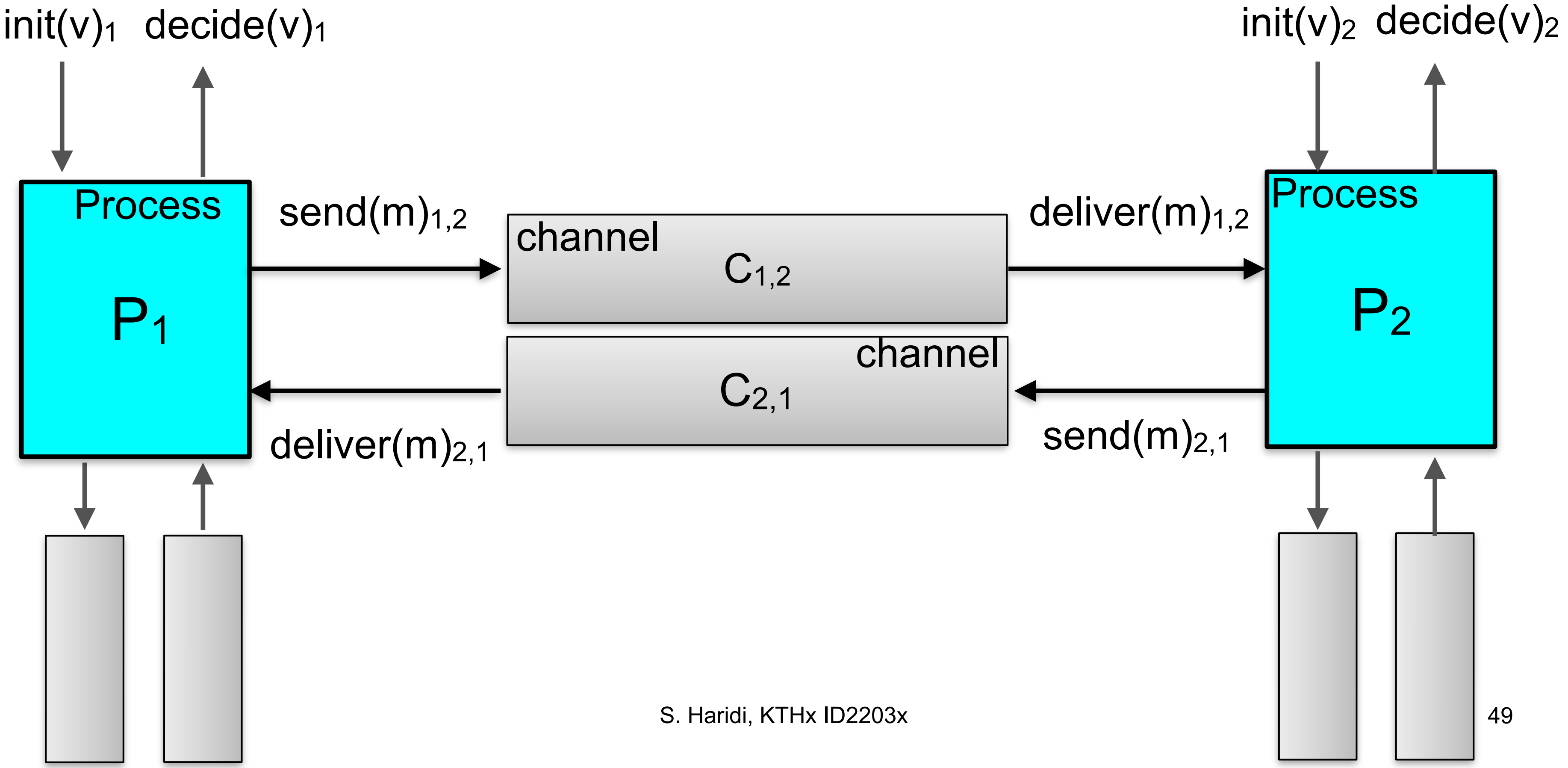
Input/output Automata

Example Composition

Distributed System Example

- In general, let $I = \{1,\ldots,n\}$
  - n process automata P_i, $i \in I$,
  - n^2 channel automata $C_{i,j}$, i and $j \in I$
- The composition automaton represents a distributed system where processes communicate through reliable FIFO channels
- The system state
  - state for each process (each a vector of values, one per process)
  - a state for each channel (each a queue of messages in transit)

Composition of channels and processes



Distributed System Example

- Transitions involve the following actions:
  - $init(v)_i$: input action, deposits a value in P_i's $val(i)$ variable
  - $send(v)_{i,j}$: output action, P_i's value $val(i)$ gets put into channel $C_{i,j}$
  - $deliver(v)_{i,j}$: output action, the first message in $C_{i,j}$ is removed and simultaneously placed into P_j's variable $val(i)$
  - $decide(v)_i$ output action at $P_{i,}$ announce current computed value

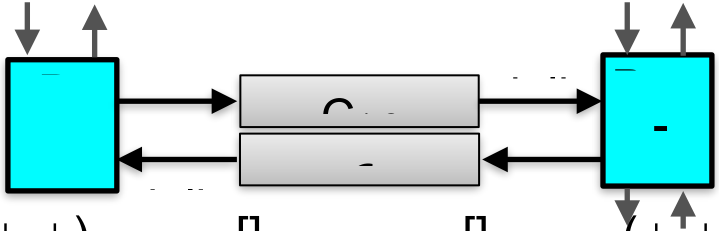
- The execution of these actions (event) defines what happens in this system

Distributed System Traces

- Sample trace, for n = 2, where the value set V is the set natural numbers N (non-negative integers) and f is addition:

- $init(2)_1$, $init(1)_2$, $send(2)_{1,2}$, $deliver(2)_{1,2}$, $send(1)_{2,1}$, $deliver(1)_{2,1}$, $init(4)_1$, $init(0)_2$, $decide(5)_1$, $decide(2)_2$

- unique system state that is reachable using this trace
  - P1 has val vector (4, 1) and P2 has val vector (2, 0),



	$(\perp,\perp)$	[]	[]	$(\perp,\perp)$
$\mathrm{init}(2)_1$,	$(2,\perp)$	[]	[]	$(\perp,\perp)$
$\mathrm{init}(1)_2$,	$(2,\perp)$	[]	[]	$(\perp,1)$
$\mathrm{send}(2)_{1,2}$,	$(2,\perp)$	[2]	[]	$(\perp,1)$
$\mathrm{deliver}(2)_{1,2}$,	$(2,\perp)$	[]	[]	$(2,1)$
$\mathrm{send}(1)_{2,1}$,	$(2,\perp)$	[]	[1]	$(2,1)$
$\mathrm{deliver}(1)_{2,1}$,	$(2,1)$	[]	[]	$(2,1)$
$\mathrm{init}(4)_1$,	$(4,1)$	[]	[]	$(2,1)$
$\mathrm{init}(0)_2$,	$(4,1)$	[]	[]	$(2,0)$

$\mathrm{decide}(5)_1$, 4+1

$\mathrm{decide}(2)_2$ 2+0

Input/output Automata

Basic Results of Automata Composition

Composition versus Components

- Execution or trace of a composition can be projected to yield executions or traces of the component automata
- Executions of component automata can be pasted together to form an execution of the composition
- Traces of component automata can be pasted together to form a trace of the composition

Similarity of executions

- The projection of component A_i in execution of E of a composed automata A, denoted $E|A_i$, is
  - the subsequence of execution E restricted to events (actions) and state of A_i
- Two executions E and F are similar w.r.t A_i if
  - $E|A_i = F|A_i$
- Two executions E and F are similar if
  - E and F are similar w.r.t every component automaton A_i

Similarity of traces

- The projection of component A_i in the trace of E of composed automata A, denoted trace(E)|A_i, is
  - the subsequence of trace(E) restricted to events of A_i
- Two traces trace(E) and trace(F) are similar w.r.t A_i if
  - $E|A_i = F|A_i$
- Two traces trace(E) and trace(F) are similar if
  - trace(E) and trace(F) are similar w.r.t every node

Projection (process view)

- Given an execution E of $A = \prod\{A_i, i \in I\}$
  - $E = s_0, a_1, s_2, \ldots$
- Projection for E on A_i, $E \mid A_i$
  - Involves deleting actions that don't belong to A_i, and the following states, and then projecting the remaining states on the A_i component
- Projection for sequence of actions β on A_i, $\beta \mid A_i$
  - Involves deleting actions that don't belong to A_i,

Distributed System Traces

- Sample trace, for n = 2, where the value set V is the set natural numbers N (non-negative integers) and f is addition:
- $init(2)_1$, $init(1)_2$, $send(2)_{1,2}$, $deliver(2)_{1,2}$, $send(1)_{2,1}$, $deliver(1)_{2,1}$, $init(4)_1$, $init(0)_2$, $decide(5)_1$, $decide(2)_2$
- unique system state that is reachable using this trace
  - P1 has val vector (4, 1) and P2 has val vector (2, 0),

Projection of Trace on P1

- Sample trace, for n = 2, where the value set V is the set natural numbers N (non-negative integers) and f is addition:
- $init(2)_1$, $init(1)_2$, $send(2)_{1,2}$, $deliver(2)_{1,2}$, $send(1)_{2,1}$, $deliver(1)_{2,1}$, $init(4)_1$, $init(0)_2$, $decide(5)_1$, $decide(2)_2$
- $init(2)_1$, $send(2)_{1,2}$, , $deliver(1)_{2,1}$, $init(4)_1$, $decide(5)_1$
- unique system state that is reachable using this trace
  - P1 has val vector (4, 1) and P2 has val vector (2, 0),

Composition versus Components

- Execution or trace of a composition projects to yield executions or traces of the component automata
- **Theorem Projection**
- Let $A = \prod\{A_i, i \in I\}$ where A_i are compatible
  - If $E \in execs(A)$, then $E \mid A_i \in execs(A_i)$ for all A_i
  - If $\beta \in traces(A)$, then $\beta \mid A_i \in traces(A_i)$ for all A_i

Composition versus Components

- Executions of component automata can be pasted together to form an execution of the composition
- Suppose E_i is an execution of A_i, β a sequence of external actions of A
- If $\beta \mid A_i$ is a trace of A_i, for all A_i, then there is an execution E of A, such that β is the trace(E) and $E_i = E \mid A_i$ for all A_i

Composition versus Components

- Traces of component automata can be pasted together to form a trace of the composition

- Suppose β a sequence of external actions of A
- If $\beta \mid A_i$ is a trace of A_i , for all A_i , then β is a trace of A

Input/output Automata

Fairness

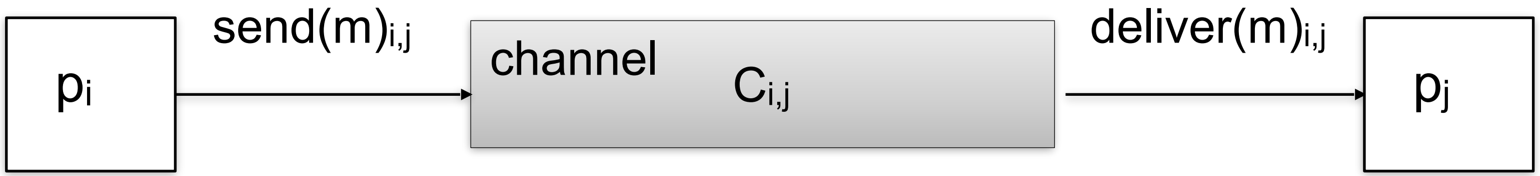
Tasks and Fairness

- Task T
  - set of of locally controlled actions
  - corresponds to a “thread of control” used to define “fair” executions
- Fairness means
  - A task that is continuously enabled gets to make a transition step
  - Needed to prove progress properties (liveness) of systems

Fairness Formally

- Formally, execution (or fragment) E of A is fair to task T if one of the following holds
  - E is finite and T is not enabled in the final state of E
  - E is infinite and contains infinitely many events in T
  - E is infinite and contains infinitely many states in which T is not enabled
- Execution of A is fair if it is fair to all tasks of A
  - fairexecs(A) is the set of fair executions of A
- Trace of A is fair if it is the trace of a fair execution of A
  - fairtraces(A) is the set of fair executions of A

Fair Executions: Channel Automaton



- Let M = {1,2}
- Three possible executions and traces

1. [λ], $send(1)_{i,j}$, [1], $deliver(1)_{i,j}$, [λ], $send(2)_{i,j}$, [2], $deliver(2)_{i,j}$, [λ]
2. $send(1)_{i,j}$, $deliver(1)_{i,j}$, $send(2)_{i,j}$, $deliver(2)_{i,j}$
3. [λ], $send(1)_{i,j}$, [1], $deliver(1)_{i,j}$, [λ], $send(2)_{i,j}$, [2]
4. $send(1)_{i,j}$, $deliver(1)_{i,j}$, $send(2)_{i,j}$
5. [λ], $send(1)_{i,j}$, [1], , $send(1)_{i,j}$, [11], , $send(1)_{i,j}$, [111], …
6. $send(1)_{i,j}$, $send(1)_{i,j}$, $send(1)_{i,j}$, …

Distributed systems examples

- Consider the fair executions of distributed system example (n processes and n^2 channels)
  - In every fair execution, every message that is sent is eventually delivered
  - In every fair execution containing at least one $init(v)_i$ event for each P_i, each process sends infinitely many messages to each other process
  - In every fair execution each process performs infinitely many decide steps

Composition versus Components

- Fair execution or trace of a composition projects to yield fair executions or traces of the component automata
- **Theorem Projection**
- Let $A = \prod\{A_i, i \in I\}$ where A_i are compatible
  - If $E \in \text{fairexecs}(A)$, then $E \mid A_i \in \text{fairexecs}(A_i)$ for all A_i
  - If $\beta \in \text{fairtraces}(A)$, then $\beta \mid A_i \in \text{fairtraces}(A_i)$ for all A_i

Composition versus Components

- Fair Executions of component automata can be pasted together to form a fair execution of the composition
- Suppose E_i is an fair execution of A_i, β a sequence of external actions of A
- If $\beta \mid A_i$ is a fair trace of A_i , for all A_i , then there is an fair execution E of A, such that β is the fairtrace(E) and E_i = $E \mid A_i$ for all A_i

Composition versus Components

- Fair traces of component automata can be pasted together to form a fair trace of the composition

- Suppose β a sequence of external actions of A
- If $\beta \mid A_i$ is a fair trace of A_i, for all A_i, then β is a fair trace of A

Input Output Automata

Trace Properties

Trace Properties

- Properties of input-output automata are formulated as properties of their fair traces
- A trace property P
  - sig(P) signature containing no internal actions
  - traces(P) a set of sequences of actions in sig(P)

Automaton A satisfied P

- Every external behavior that can be produced by A is permitted by property P

- A satisfies a trace property P can mean either
  - extsig(A) = sig(P) and traces(A) $\subseteq$ traces(P), or
  - extsig(A) = sig(P) and fairtraces(A) $\subseteq$ traces(P)

Example

- Automata A and trace property P has
  - {0} as input set
  - {0,1,2} as output set
- traces(P)
  - is the set of all sequences of {0,1,2} that include at least one 1
- A has a task that always output 1
- fairtraces(A) $\subseteq$ traces(P)
- traces(A) $\not\subseteq$ traces(P)
  - Empty sequence is in traces(A)

Safety properties

- A safety property P states that some particular "bad" thing never happens in any trace
- A trace property P is a safety property if
  - traces(P) is nonempty
  - if $\beta \in$ traces(P) then every finite prefix of β is in traces(P)
    - if nothing bad happens in β then nothing bad happens in a prefix of β
  - if $\beta 1$, $\beta 2$,… is an infinite sequence of finite traces in traces(P) where each β_i is a prefix of β_{i+1} then the limit β is also in traces(P)
    - if something bad happens in (infinite) β then a bad event happens in a finite prefix

Example

- A trace property P has
  - init(v): $v \in V$ as input set
  - decide(v): $v \in V$ as output set
- traces(P)
  - is the set of all sequences of init(v) and decide(v) where no decide(v) occurs without a preceding init(v)

Liveness properties

- Informally a liveness property is saying that some particular "good" thing eventually happens
- A trace property P is a liveness property if
  - every finite sequence over sig(P) has some extension that is in traces(P)

Example

- A trace property P has
  - init(v): $v \in V$ as input set
  - decide(v): $v \in V$ as output set
- traces(P)
  - is the set of all sequences of init(v) and decide(v) where for every init(v) event in a sequence there is a decide(v) event later in the sequence

Relating safety and liveness

- Two important results
- **Theorems**
  - If P is both a safety property and a liveness property, then P is the set of all sequences of actions in sig(P)

  - If P is an arbitrary trace property with traces(P) $\neq \emptyset$, then there exist a safety property S and a liveness property L such that
    - traces(P) = traces(S) $\cap$ traces(L)